Sound and Complete Mutation-Based Program Repair

Bat-Chen Rothenberg and Orna Grumberg

CS Department, Technion, Israel
{batg,orna}@cs.technion.ac.il

Abstract. This work presents a novel approach for automatically repairing an erroneous program with respect to a given set of assertions. Programs are repaired using a predefined set of mutations. We refer to a bounded notion of correctness, even though, for a large enough bound all returned programs are fully correct. To ensure no changes are made to the original program unless necessary, if a program can be repaired by applying a set of mutations Mut, then no superset of Mut is later considered. Programs are checked in increasing number of mutations, and every minimal repaired program is returned as soon as found.

We impose no assumptions on the number of erroneous locations in the program, yet we are able to guarantee soundness and completeness. That is, we assure that a program is returned iff it is minimal and bounded correct.

Searching the space of mutated programs is reduced to searching unsatisfiable sets of constraints, which is performed efficiently using a sophisticated cooperation between SAT and SMT solvers. Similarities between mutated programs are exploited in a new way, by using both the SAT and the SMT solvers incrementally.

We implemented a prototype of our algorithm, compared it with a state-of-the-art repair tool and got very encouraging results.

1 Introduction

In the process of software production and maintenance, much effort and many resources are invested in order to ensure that the product is as bug free as possible. Manual bug repair is time-consuming and requires close acquaintance with the checked program. Therefore, there is a great need for tools performing automated program repair. In recent years, there has been much progress in this field (e.g., [14, 23, 22, 6, 19, 12]).

In previous work, the presented motivation for the development of program repair tools is to enable the automatic repair of real-world bugs found in large-scale software projects. As a result, existing tools for automated repair aim at being scalable and are targeted for the type of bugs found in deployed software.

We have designed our algorithm with a different goal in mind. In our opinion, automatic repair can be equally or even more useful when applied in the earlier stages of development, before any manual effort was invested in debugging at all.

This is because, in our view, it is precisely the initial debugging work that could benefit the most from this automation, since it involves relatively simple bugs being fixed manually by the programmer. For these early development stages, as well as for millions of independent programmers working on small pieces of code, even a non-scalable automatic repair method can help save a lot of time and avoid much frustration.

Our vision is to have a fast, easy-to-use program repair tool, which programmers can run routinely. Ideally, programmers will run the tool immediately after making changes to the program, before any manual effort was invested in debugging at all. Then, if the program contains an assertion violation, the chosen course of action will be determined by the tool's result. If the tool returns one or more possible repairs, those are guaranteed to suppress all assertion violations and thus may be safely applied to the program. If the tool does not return any possible repairs, the programmer can be sure that the problem can not be solved using changes within the search space of the tool. In the later case, though manual debugging will still be needed, knowing what will not solve the problem might give the programmer a head start.

In this work, we take a step forward towards accomplishing this vision, presenting a novel algorithm for automatically repairing a program with respect to a given set of assertions. We use a bounded notion of correctness. That is, for a given bound b, we consider only *bounded computations*, along which each loop in the program is performed at most b times and each recursive call is inlined at most b times. We say that a program is *repaired* if whenever a bounded computation reaches an assertion, the assertion is evaluated to true. Our repair method is *sound*, meaning that every returned program is repaired (i.e., no violation occurs in it up to the given bound). Just like Bounded Model Checking, this increases our confidence in the returned program.

Our programs are repaired using a predefined set of mutations, applied to expressions in conditionals and assignments (e.g. replacing a $+$ operator by a $-$), as was shown useful in previous work [4, 27, 5]. We impose no assumptions on the number of mutations needed to repair the program and are able to produce repairs involving multiple buggy locations, possibly co-dependent. To make sure that our suggested repairs are as close to the original program as possible, the repaired programs are examined and returned in increasing number of mutations. In addition, only *minimal* sets of mutations are taken into account. That is, if a program can be repaired by applying a set of mutations Mut, then no superset of Mut is later considered. Intuitively, this is our way to make sure all changes made to the program by a certain repair are indeed necessary. Our method is *complete* in the sense of returning *all* minimal sets of mutations that create a repaired program. Specifically, if no repair is found, one can conclude that the given set of mutations is not enough to repair the program. Furthermore, we show that for large enough bound, all returned programs are (unbounded) fully correct.

Note that, the choice to use mutations for repair makes the search space small enough to enable us to have completeness at an affordable cost, yet it is expressive enough to repair meaningful bugs (especially those present in earlier stages of development).

Our algorithm is based on the translation of the program into a set of SMT constraints which is satisfiable (i.e., the conjunction of constraints in it is satisfiable) iff the program contains an assertion violation. This was originally done for the purpose of bounded model checking in [1][1]. Our key observation is that mutating an expression in the program corresponds to replacing a constraint in the set of constraints encoding the program. Thus, searching the space of mutated programs is reduced to searching unsatisfiable sets of constraints. The latter can be performed efficiently using a sophisticated cooperation between SAT and SMT solvers, as was done in [16] for the purpose of finding minimal unsatisfiable cores.

The SAT solver is used to restrict the search space of mutated programs to only those obtained by a minimal mutation set and the SMT solver verifies whether a mutated program is indeed correct. Both the SAT solver and the SMT solver are used incrementally, which means that learned information is passed between successive calls, resulting in big savings in terms of resources used. Using an SMT solver incrementally constitutes a novel way to exploit information learned while checking the correctness of one program for the process of checking correctness of another program. Note, that if the programs are similar, their encoding as sets of SMT constraints will also be similar (due to our observation presented above), resulting in bigger savings when using incremental SMT. This is another important contribution of this paper.

We implemented a prototype of our algorithm for C programs, compared it with the methods of [11, 12] and got very encouraging results.
To summarize, the main contributions of our work are:

- We propose a novel *sound and complete* algorithm which returns *all* minimal repaired programs.
- The returned programs are proved to be bounded correct. However, we show that for a large enough bound, all returned programs are fully correct and all minimal fully correct programs are returned.
- We develop an efficient implementation of the algorithm, based on sophisticated cooperation between SAT and SMT solvers, both used incrementally.

1.1 Related work

Several repair methods follow a test-based "generate and validate" approach. They iteratively select a candidate from the repair search space and check its validity by running all tests in the test suite against it. Examples are GenProg [14, 13], TrpAutoRepair [25], AE [31], RSRepair [26] and the more recent SPR [19]. PAR [9], Monperrus and Martinez [20] and Prophet [18] suggest to use information learned from successful human repairs to extract and prioritize repair

[1] To be precise, [1] first translates the program into a bit-vector formula and then further translates it into a propositional formula. Here, we only use the first part of the translation

actions suitable for the suspected location of the error. Similarily, CodePhage [28] directly transfers pieces of code from correct donor applications to buggy recipient ones. AutoFix-E [30] and AutoFix [24] also use location based repair actions, but require programs to be equipped with contracts.

SemFix [23], DirectFix [21] and Angelix [22] use symbolic execution to infer a repair constraint and synthesize a repair based on it. Nopol [6] also uses synthesis, but only deals with buggy if conditions and missing pre conditions. [10] uses deductive synthesis and is based on pre and post conditions, rather than tests alone. [8] and [29] describe systems using automata and use LTL specifications for repair.

Mutation based program repair (where the term "mutation" has the same meaning as in this work) was previously done in [27] and [5]. Both use a test suite as the only specification and focus their efforts on efficient error localization. We, on the other hand, use a formal specification and have no use of localization, since we have to consider all locations in order to guarantee completeness. Also, we allow the repair of multiple expressions, whereas both methods assume a single fault ([5] mentions a possible extension to multiple faults, but this is not a part of the described method).

Finally, he methods of [11, 12] are similar to ours in that they work on C programs equipped with assertions (or test suites) and assume faulty expressions. The differences are that they use program analysis based on a finite number of inputs each time, while we use incremental SMT solving that allows reuse of information. Also, they use templates (e.g. a linear combination of variables) for repair, while we use mutations and are able to guarantee completeness. We provide a comparison of performance results between our method and theirs in sec. 6.

2 Preliminaries

Program correctness For our purposes, a *program* is a sequential program composed of standard commands: assignments, conditionals, loops and function calls. Each command is located at a certain *program location* l_i, and all commands are defined over the set of program variables X.

In addition to the standard commands, a program may contain assumptions and assertions, which are commands that help the user specify the desired behavior. *Assumptions* (resp., *assertions*) are commands of the form assume(e) (resp., assert(e)), where e is a boolean expression over X. An assertion assert(e) at location l_i, specifies that the user expects e to evaluate to true whenever control reaches l_i, in all program runs. If e evaluates to true every time control reaches l_i during a run r, we say the assertion *holds* for r. Otherwise, the assertion is *violated*. Once an assertion in the program is violated, the program terminates (this early termination indicates an error has occurred and is usually preceded by an error message explaining what went wrong). An assumption assume(e) at location l_i, specifies that every run reaching l_i with e evaluated to false is terminated. Unlike before, this early termination is not an indication that something went wrong, but simply that the user does not want to consider the rest of this run when checking correctness. For example, if a function f gets as input an integer n, but the user assumes it will only be called with $n \geq 2$, an assumption $assume(n \geq 2)$ can be inserted at the beginning of the function to make sure all runs in which this function is called inappropriately will be truncated.

Def. 1 (correct program)**.** *A program is correct if all assertions in it hold in all runs.*

For a program P and an integer b, a *b-run* of P is a run of P that goes through each loop at most b times and has a recursion depth of at most b (i.e., the depth of the call stack is at most b during the entire run).

Def. 2 (b-correct program)**.** *Let b be an integer. A program is b-correct if all assertions in it hold in all b-runs.*

Our repair method aims at finding programs which are b-correct, therefore we use the term *repaired program* as a notation for a b-correct program.

2.1 Incremental SAT and SMT solving

A SAT solver is a decision procedure for deciding the satisfiability of a propositional formula. Formulas are usually in conjunction normal form (CNF) and can also be seen as a set of clauses. *Incremental SAT solving* is a general name for a set of techniques aimed at improving the SAT solver's performance when called repeatedly for similar formulas (i.e., similar sets of clauses). The basic principal behind these techniques is to save running time by retaining information learned by the SAT solver between calls.

An SMT solver (where SMT stands for satisfiability modulo theories), is another kind of decision procedure of much recent interest. It decides the satisfiability of a formula expressed in first order logic (FOL), where the interpretation of some symbols is constrained by a background theory (for more details see [3]). Examples of commonly used theories are the theory of linear arithmetic over integers and the theory of arrays. Just like a CNF formula can be seen as a set of clauses, an SMT formula can be seen as a set of constraints in the theory (referred to as *SMT constraints*).

Similarly to SAT solving, incremental techniques can be applied to SMT solving as well. For this to be useful, an SMT formula φ is usually instrumented with boolean variables called *guard variables*. The instrumentation of a formula φ is done as follows: each constraint $c_i \in \varphi$ is replaced by the constraint $x_i \rightarrow c_i$, where x_i is a fresh boolean variable. As a result, the new constraint can easily be satisfied by setting x_i to false. Guard variables are conjuncted with φ and are used as *assumptions*, passed to an incremental SMT solver. They have the effect of canceling out a subset of constraints. For example, if $\varphi = c_1 \wedge c_2$, after instrumentation we get the formula $\varphi' = (x_1 \rightarrow c_1) \wedge (x_2 \rightarrow c_2)$. Calling an incremental SMT solver on φ' with the set of assumptions $\{x_1\}$ causes the SMT solver to check the satisfiability of $\varphi' \wedge x_1$, which essentially disables the constraint c_2. That is, because nothing prevents x_2 from being set to false, and x_1 must be set to true, checking satisfiability of φ' is reduced to checking satisfiability of c_1.

Boolean cardinality constraints Boolean cardinality constraints are constraints of the form $\sum_{i=1}^{n} l_i \leq k$, where l_i is a literal assigned the value 1 if true and 0 if false, and k is an integer constant. For readability, we will refer to these constraints using the notation AtMost$(\{l_1, .., l_n\}, k)$, also used in [17], in order to remind the reader of their intuitive meaning: require that at most k of these literals get the value true. Similarly, the notation AtLeast$(\{l_1, .., l_n\}, k)$, denotes the constraint $\sum_{i=1}^{n} l_i \geq k$. For our implementation we used Minicard [15], which is a SAT-solver designed to perform well on instances containing cardinality constraints.

3 Our approach

In this section we fix a bound b and refer to repaired programs which are b-correct. Figure 1 presents an overview of our repair system. It is composed of three units: the translation unit, the mutation unit and the repair unit.

The initial processing is done in the translation unit. The translation unit translates the input program into two sets of SMT constraints: S_{hard}, encoding parts of the program which cannot be changed (e.g. assertions), and S_{soft}. Then, the mutation unit constructs for each constraint c_i in S_{soft} a set of alternative constraints S_i, by applying mutations to c_i. Finally, the repair unit searches for all sets of constraints encoding minimal repaired programs (where minimality will be defined with respect to the set of mutations used). In the rest of the section we explain in detail how each unit works.

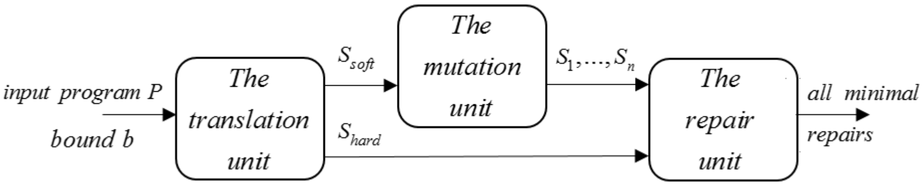


Fig. 1: Overview of the repair system

3.1 The translation unit

The translation unit is the first step of the process. It gets an input program and an integer bound b and converts the input program into a set of SMT constraints s.t. the program is b-correct iff the set of constraints is unsatisfiable (i.e the conjunction of all constraints in it is unsatisfiable).

Before the set of constraints is constructed, the program undergoes three transformations: simplification, unwinding, and conversion to static single assignment (SSA) form. This transformations are taken from [1], but we present them here because the details are important in order to understand our method.

To explain the different transformations we will use the example presented in Figure 2. Figure 2a presents a C function named sum, which gets as input an integer n and is supposed to return $\sum_{1}^{n} i$. But, being used to 0-based counting, the programmer made a mistake in line 3, by initializing i to 0 instead of 1 and checking $i < n$ instead of $i <= n$. The assertion in line 6 specifies that the result should always be calculated according to the formula $\frac{n \cdot (n+1)}{2}$, which is the correct sum calculated using the formula for a sum of an arithmetic progression.

We will now go over each transformation and explain its role shortly, using the described example.

Simplification Figure 2b shows the result of applying simplification to the program in Figure 2a. Complex constructs are replaced with simpler ones (for example, the for loop was replaced with a while loop). More importantly, all conditions are assigned to auxiliary boolean variables (g in the example). Note that after this step, all original program expressions are right-hand-sides of assignments.

Unwinding Figure 2c shows the result of applying unwinding for $b = 2$ to the program in Figure 2b. The loop is unwound b times by duplicating the loop body b times, where each copy is guarded using an if statement that uses the same condition as the loop statement (lines 5-15). Inside the innermost copy, an assume statement is inserted with the negation of the condition (line 13), to specify we do not want to consider runs going through the loop more than b times.[2] Function calls are inlined, with recursive calls treated similarly to loops (inserted up to a depth of b).

Conversion to SSA form The program is converted to SSA form (which means each variable is assigned only once). Figure 2d shows the result of converting the program in Figure 2c to SSA form. All variables are replaced with indexed variables, and whenever a variable appears as the left-hand-side of an assignment, its index is increased by 1. If a variable x is assigned inside a conditional statement and is used after the statement, an assignment is inserted straight after the conditional statement to determine which copy of x should be used. For example, lines 16-17 determine the updated value of *sum* after the nested if statements, according to $g1$ and $g2$. We refer to this type of assignments as Φ-assignments.

After the above transformations, conversion to a set of SMT constraints S is straightforward. An assignment $x = e$ is converted to the constraint $x = e$, an

[2] In [1] an assertion was inserted and not an assume. Since we fix the program with respect to all assertions in it, we need this to be an assume and not an assert, because we do not want to refer to unbounded runs as bugs.

$assume(e)$ is converted to the constraint e and an $assert(e)$ is converted to the constraint $\neg e$.[3] Shortly, we say that a constraint *encodes* a statement.

In the next step, the mutation unit will apply mutations independently to every constraint passed to it. The problem is that, due to unwinding, all statements which are part of a loop (as the loop condition or in the loop body) are encoded using more than one constraint in S. This is of course undesirable, because we do not want constraints encoding the same statement to be mutated using different mutations. To avoid this, if a statement s is encoded using the constraints $c_1, ..., c_t \in S$ (where $t > 1$), we remove $c_1, ..., c_t$ from S, and add instead one complex constraint, $\bigwedge_{i=1}^{t} c_i$. Note that this has no effect on the satisfiability of S (which is determined by the conjunction of all constraints in S anyway).

As a final step, the modified set S is partitioned into two sets: S_{soft}, containing all constraints encoding statements subject to repair (i.e statements containing original program expressions), and S_{hard}, containing the rest (constraints encoding negated assertions, assumptions and Φ-assignments). Note that since we made sure all original program expressions are right-hand-sides of assignments using simplification, we can be sure all constraints in S_{soft} are of the form $(x = e)$ (where x is an SSA variable and e is an expression), or of the form $(c_1 \wedge c_2, ..., \wedge c_n)$ where each c_i is of the form $(x = e)$. Furthermore, we can be sure all program statements which are subject to repair are encoded using a single constraint and vice versa, and thus the size of S_{soft} will always be the same as the number of original program expressions (regardless of the bound b).

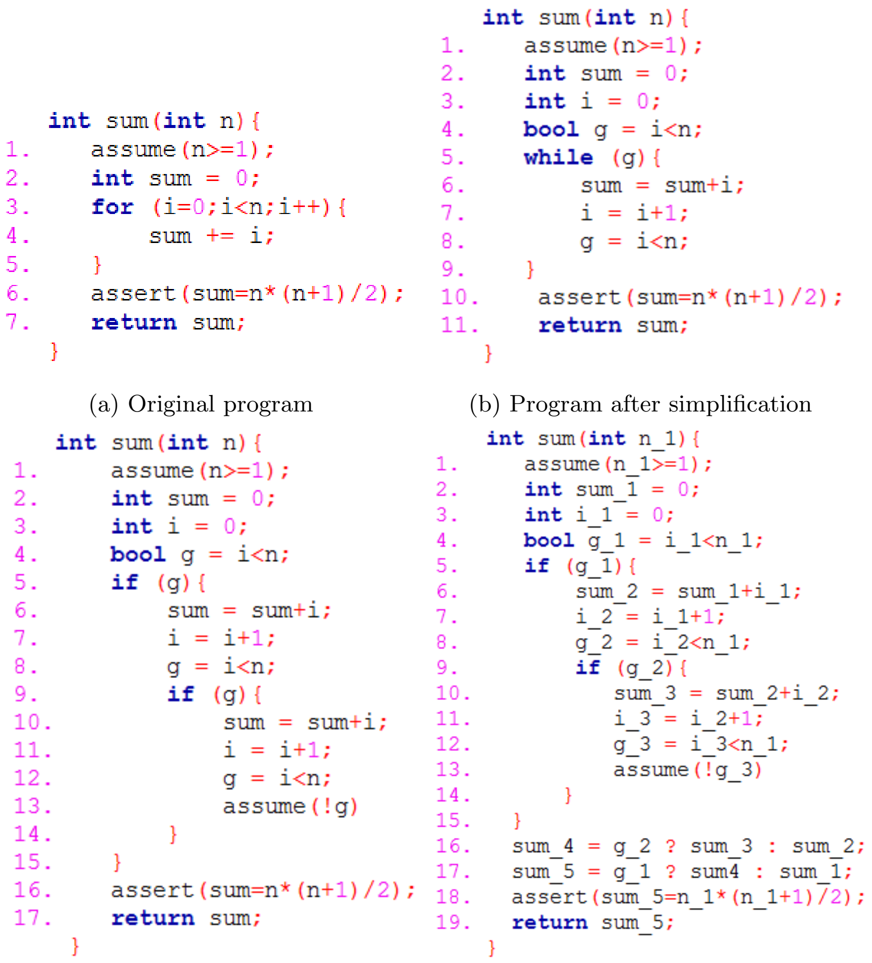
3.2 The mutation unit

We assume the program is incorrect because it contains one or more faulty expressions, and we try to repair it by applying mutations to program expressions. A *mutation* can be any function mapping a program expression to another program expression of the same type. Examples of mutations include replacing an operator by a similar one (e.g., $\leq$ by $<$) and applying constant manipulations (e.g., replacing a constant by 0). The mutation unit is the component in charge of applying the mutations. In fact, as described in Fig. 1, the mutations are not applied directly on the program, but on constraints encoding the program, received from the translation unit.

As explained in Section 3.1, the constraints in the input set, S_{soft}, can be single assignment constraints or multiple assignments constraints. Formally, given a mutation M, and a single assignment constraint $(x = e)$, $M(x = e)$ is the constraint $(x = M(e))$. For a multiple assignment constraint $c = (c_1 \wedge c_2 \wedge ... \wedge c_t)$, $M(c)$ is the constraint $(M(c_1) \wedge M(c_2) \wedge ... \wedge M(c_t))$.

The mutation unit maintains a fixed list of possible mutations, $M_1, M_2, ..., M_m$. For each $c_i \in S_{soft}$ $(1 \leq i \leq n)$ all the mutations are applied and the set

[3] Assertions are negated because we want a satisfying assignment to the set of constraints to represent a violation of the assertion. If multiple assertions exist in the code, the disjunction of their negations is added as a constraint.



```
int sum(int n){
    assume(n>=1);
    int sum = 0;
    for (i=0;i<n;i++){
        sum += i;
    }
    assert(sum=n*(n+1)/2);
    return sum;
}
```

(a) Original program

```
int sum(int n){
    assume(n>=1);
    int sum = 0;
    int i = 0;
    bool g = i<n;
    while (g){
        sum = sum+i;
        i = i+1;
        g = i<n;
    }
    assert(sum=n*(n+1)/2);
    return sum;
}
```

(b) Program after simplification

```
int sum(int n){
    assume(n>=1);
    int sum = 0;
    int i = 0;
    bool g = i<n;
    if (g){
        sum = sum+i;
        i = i+1;
        g = i<n;
        if (g){
            sum = sum+i;
            i = i+1;
            g = i<n;
            assume(!g)
        }
    }
    assert(sum=n*(n+1)/2);
    return sum;
}
```

(c) Program after unwinding for $b = 2$

```
int sum(int n_1){
    assume(n_1>=1);
    int sum_1 = 0;
    int i_1 = 0;
    bool g_1 = i_1<n_1;
    if (g_1){
        sum_2 = sum_1+i_1;
        i_2 = i_1+1;
        g_2 = i_2<n_1;
        if (g_2){
            sum_3 = sum_2+i_2;
            i_3 = i_2+1;
            g_3 = i_3<n_1;
            assume(!g_3)
        }
    }
    sum_4 = g_2 ? sum_3 : sum_2;
    sum_5 = g_1 ? sum4 : sum_1;
    assert(sum_5=n_1*(n_1+1)/2);
    return sum_5;
}
```

(d) Program after conversion to SSA

Fig. 2: Example of program transformations during translation

$S_i = \{c_i, M_1(c_i), ..., M_m(c_i)\}$ is created.[4] Note that the set S_i contains the original constraint c_i, so leaving a statement intact is always an option. Finally, the sets $S_1, ..., S_n$ are passed on to the repair unit, which uses them to search for a repair.

3.3 The repair unit

Basic terms and definitions The input to the repair unit is a set of "hard constraints", S_{hard}, encoding the parts of the program which can not be changed, and n disjoint sets of "soft constraints", $S_1, ..., S_n$, corresponding to n program locations where a possible fault may occur. Every set S_i contains one special constraint, c^i_o, encoding the original statement in line i, referred to as the *original constraint*. The rest of the constraints in S_i encode possible replacements for line i, obtained by applying mutations to the expression in the original statement.

Intuitively, the goal of the repair unit is to construct a repaired program by choosing one constraint from each S_i. Formally, we define a *selection vector* (**sv**) $[c_1, ..., c_n]$ as a vector of constraints where c_i is taken from S_i for all $1 \leq i \leq n$. Recall that constraints in S_i encode different statements for line i, therefore choosing a specific constraint from each S_i can be seen as choosing a statement to appear in each line, i.e choosing a mutated program. Thus, each selection vector *encodes* a program. We are interested in selection vectors encoding repaired or correct programs. This leads to the following definitions.

Def. 3 (**Rsv**,**Csv**). *A selection vector is repaired, denoted* **Rsv**, *if it encodes a repaired program. A selection vector is correct, denoted* **Csv**, *if it encodes a correct program.*

Though (bounded) correctness is essential for repair, it is not enough. We would also like for the repair to be "minimal", in the sense that no changes are made unless necessary. For example, if a program can be repaired by applying a certain mutation to line number 2, we are not interested in a repair suggesting to additionally mutate line number 3, even if it makes the program repaired. To capture this intuition we define a partial order between constraints and between selection vectors.

Def. 4 ($\sqsubseteq$ partial order between constraints)**.** *Let* $c^1_i, c^2_i \in S_i$. $c^1_i \sqsubseteq c^2_i$ *if* $c^1_i = c^i_o$ *and* $c^2_i \neq c^i_o$ *(i.e., only* c^2_i *encodes a change to line* i*), or if* $c^1_i = c^2_i$ *(i.e., both encode the same statement for line* i*).*

Def. 5 ($\sqsubseteq$ partial order between **sv**s)**.** *Let* $v_1 = [c^1_1, ..., c^1_n], v_2 = [c^2_1, ..., c^2_n]$ *be selection vectors.* $v_1 \sqsubseteq v_2$ *if for all* $1 \leq i \leq n$ $c^1_i \sqsubseteq c^2_i$.

Def. 6 (**mRsv**,**mCsv**)**.** *A repaired selection vector* v *is minimal repaired, denoted* **mRsv**, *if there is no* v' *s.t.* $v' \neq v$, v' *is a repaired selection vector and* $v' \sqsubseteq v$.

[4] This is a simplification made for ease of presentation. In practice, we might not be able to (or not want to) apply all mutations to all constraints. The choice of mutations to use may depend on the expression's type and/or its complexity.

A correct selection vector v is minimal correct, denoted **mCsv**, *if there is no v' s.t. $v' \neq v$, v' is a correct selection vector and $v' \sqsubseteq v$.*

Finally, it makes sense to prefer repairs involving as few statements as possible, because those are more likely to satisfy the user. For example, if the program can be repaired by mutating line 1 and also by mutating lines 2 and 3, the first repair is preferable. This intuition is formalized using the following definition:

Def. 7 (size)**.** *Let v be a selection vector. The size of v, denoted size(v), is* $|\{i | 1 \leq i \leq n, v[i] \neq c_o^i\}|$.

In other words, size(v) is the number of mutated lines in the program encoded by v. Thus, the repair unit should only look for minimal repaired selection vectors, and amongst them prefer those with smaller size. In what follows, we present an algorithm that computes *all* minimal repaired selection vectors (**mRsv**s), and produces results in increasing size over time.

4 Algorithm AllRepair for the repair unit

4.1 Outline of the Algorithm

Figure 3 presents the general outline of our algorithm. Overall, the algorithm goes over the search space of all **sv**s, in increasing size order. This order is enforced using the variable k, which limits the allowed size of the searched **sv**s (k is initially 1 and grows over time)[5]. Once the search reaches an **sv** v, we say v has been *explored* (until then, v is *unexplored*). The algorithm is divided into two repeating phases:

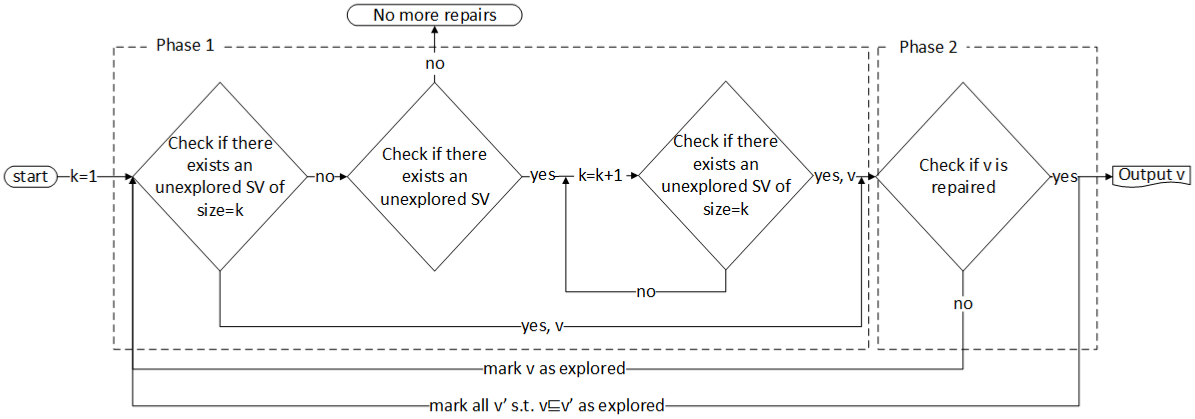

No more repairs
Phase 1
no
Phase 2
start
k=1
Check if there exists an unexplored SV of size=k
no
Check if there exists an unexplored SV
yes
k=k+1
Check if there exists an unexplored SV of size=k
yes, v
Check if v is repaired
yes
Output v
no
no
yes, v
mark v as explored
mark all v' s.t. v⊑v' as explored

Fig. 3: Outline of Algorithm **AllRepair**

[5] k is not to be confused with the unwinding bound b, which is fixed at this point.

Phase 1 is responsible for finding the next unexplored **sv**. First, it looks for an unexplored **sv** of size k. If one exists, it is passed on to Phase 2. Otherwise, it checks if there exist any unexplored **sv**s left at all. If not, the search is over and the procedure ends. Otherwise, k is repeatedly increased by one until an unexplored **sv** v of size k is found (v must be found for some k since we know an unexplored **sv** exists). Once found, v is passed on to Phase 2.

Phase 2 gets as input an unexplored **sv** v. First, it checks if v is repaired, that is, if v is b-correct. If it is, v is returned as a possible repair. In addition, if v is repaired, Phase 2 marks not only v as explored, but also every **sv** v' s.t. $v \sqsubseteq v'$. This is done in order to make sure that we will not waste time exploring v' in the future, since it is necessarily not minimal. If v is not repaired, then only v is marked as explored.

4.2 Algorithm AllRepair in detail

The pseudo-code of algorithm **AllRepair** is presented in Figure 4. This algorithm follows the general outline presented before, where an incremental SAT-solver with cardinality constraints is used for the implementation of Phase 1, and an incremental SMT-solver is used for the implementation of Phase 2. Note that, we are interested in the *satisfying assignments* returned by the SAT solver and in the *unsatisfiable instances* returned by the SMT solver. The former represent **sv**s of desired sizes while the latter represent repaired programs.

The description below is strongly based on the background given in Section 2.1. The first step is to instrument all constraints in $S_1, \ldots, S_n$ with guard variables. This is done using a function call in line 2, and the results are the sets of instrumented constraints, $S'_1, \ldots, S'_n$ (where $S'_i = \{x_j \to c_j \mid \text{for every } c_j \in S_i\}$) and the sets of fresh guard variables used to guard the constraints in each set, $V_1, \ldots, V_n$ (where V_i contains the variables x_j used to guard constraints in S_i). This instrumentation serves us in building both the SMT formula τ and the boolean formula φ (passed to the SAT-solver).

Next, in lines 3-6, τ is initialized to the conjunction of all constraints in S_{hard} and all the instrumented constraints. Notice that this will enable us to determine which of the soft constraints will be considered in each call to the SMT solver, by using their guard variables as assumptions (while hard constraints will be considered in all calls, regardless of the assumptions).

The boolean formula φ is initialized in lines 7-11. The boolean variables composing this formula are the guard variables $V_1, ..., V_n$, and therefore every satisfying assignment of it can be seen as a subset of guard variables (those assigned true by the assignment). We would like every satisfying assignment to be not just any subset of guard variables, but one consistent with the definition of an **sv**, i.e., a subset that contains exactly one selector variable from each V_i. Lines 9-10 add to φ the necessary constraints to enforce this. From now on, we will say that satisfying assignments returned by the SAT-solver *represent* **sv**s.

Next, we would like to be able to add an upper bound on the size of represented **sv**s. For this purpose, we define an additional formula, φ_k. In order to construct φ_k, we first need to identify which guard variables guard the original

```
 1: function AllRepair(Input: $S_{hard}, S_1, ..., S_n$,  Output: All mRsvs)
 2:     $S'_1, ..., S'_n, V_1, ..., V_n \leftarrow$ AddSelVars($S_1, ..., S_n$)
 3:     $\tau \leftarrow true$                                  ▷ initialization of SMT formula
 4:     for $c \in S_{hard} \cup S'_1 \cup ... \cup S'_n$ do
 5:         $\tau \leftarrow \tau \wedge c$
 6:     end for
 7:     $\varphi \leftarrow true$                               ▷ initialization of boolean formula
 8:     for $1 \leq i \leq n$ do
 9:         $\varphi \leftarrow \varphi \wedge$AtMost($V_i$,1)            ▷ choose at most one statement per line
10:         $\varphi \leftarrow \varphi \wedge (\bigvee_{v \in V_i} v)$          ▷ choose at least one statement per line
11:     end for
12:     $V_o \leftarrow$ GetSelVarsOfOriginal($V_1, ..., V_n$)
13:     $k \leftarrow 1$
14:     while true do
15:         $\varphi_k \leftarrow \varphi \wedge AtLeast(V_o, n-k)$
16:         $satRes, V \leftarrow SAT(\varphi_k)$
17:         if $satRes$ is unsat then
18:             if $\neg SAT(\varphi)$ then                      ▷ No more svs to explore
19:                 return
20:             end if
21:             repeat
22:                 $k \leftarrow k+1$
23:                 $\varphi_k \leftarrow \varphi \wedge AtLeast(V_o, n-k)$
24:                 $satRes, V \leftarrow SAT(\varphi_k)$
25:             until $satRes$ is sat
26:         end if
27:         $smtRes \leftarrow$ IncrementalSMT($\tau, V$)        ▷ at this point $V$ has been assigned
28:         if $smtRes$ is SAT then
29:             $\varphi_{\text{Block}} \leftarrow$ BlockUnrepairedsv($V$)
30:         else
31:             output Getsv($V, S'_1, ..., S'_n$)
32:             $\varphi_{\text{Block}} \leftarrow$ BlockRepairedsv($V$)
33:         end if
34:         $\varphi \leftarrow \varphi \wedge \varphi_{\text{Block}}$                 ▷ $\varphi$ includes new blocking; $k$ is not changed
35:     end while
36: end function
```

Fig. 4: Algorithm **AllRepair** for finding all **mRsv**s

constraints. This is done in the function call in line 12, and the result is stored in V_o.

Lines 13-26 essentially implement Phase 1 of the outline in Figure 3. k is initialized to 1 (line 13) and the iterative repetition of the two phases begins. First, φ_k is set to the conjunction of φ and the clause AtLeast$(V_o, n-k)$ (line 15). That is, in φ_k we additionally require that at least $n-k$ variables from V_o get the value true. This essentially means that every satisfying assignment to φ_k now represents an **sv** of size at most k.

Next, we check whether there exists an unexplored **sv** of size at most k by sending φ_k to the SAT solver (line 16). The satisfiability result (sat/unsat) is saved into *satRes*, and if the result is sat, V gets the set of all variables assigned true by the satisfying assignment. If the result is unsat, we check whether there exists an unexplored **sv** (without limitation on size) by sending φ to the SAT solver (line 18). If the result is unsat, the algorithm ends (line 19). Otherwise, we repeatedly increase k by one and resend φ_k to the SAT solver, until the result is sat (lines 21-25).

Phase 2 begins in line 27, by calling the function IncrementalSMT(τ, V), which checks the satisfiability of τ with all variables in V passed as assumptions. This is in fact equivalent to checking the satisfiabillity of the conjunction of all constraints in S_{hard} and all soft constraints guarded by variables in V (since all other constraints can be easily satisfied by setting their guard variables to false). Note that this formula is unsat iff the **sv** represented by V (i.e., the constraints guarded by variables in V) is an **Rsv**. Therefore, if the result is sat, we create a blocking clause φ_{Block} for the case in which V represents an **sv** that is not repaired (line 29). The blocking clause in this case is simply $\bigvee_{v \in V} \neg v$ (i.e only V is blocked). If the result is unsat, we translate V into the represented **sv** and return it as a possible repair (line 31). The blocking clause we add in this case (line 32) is $\bigvee_{v \in V \setminus V_o} \neg v$, which requires that the same set of mutations will never appear as a subset of any future set of mutations. This way we block not only V but also every V' for which $v \sqsubseteq v'$ (where v, v' are the **sv**s represented by V, V', respectively).

5 Soundness and completeness of Algorithm AllRepair

In this section we analyze our algorithm. We show that it is *sound*, that is, every returned **sv** is minimal repaired, and that it is *complete* in the sense that every minimal repaired **sv** is eventually returned.

Clearly, the algorithm returns all **mRsv**s, because we go over all **sv**s and only mark an **sv** as explored if it is returned (as repaired), if it is not repaired, or if it is not minimal. Also, all **sv**s returned by the algorithm are **mRsv**s, because every returned **sv** is repaired (it is explicitly checked), and is minimal repaired because otherwise it would have been marked as explored by another **sv** in a previous iteration. Thus, the following theorem holds:

Theorem 8 (Correctness of **AllRepair**)**.** *Our algorithm is sound and complete. That is, every* **sv** *v returned by our algorithm is an* **mRsv** *and every* **mRsv** *v is returned by our algorithm at some point.*

5.1 Extension to full Correctness

We now analyze the soundness and completeness of our algorithm with respect to full (unbounded) correctness. We show that there is a bound B for which the notion of B-correctness is equivalent to the notion of correctness.

We first notice that since the set of mutations we consider is finite, so is the set of mutated programs PG. For each $P \in PG$, if it is not correct then it has a b-run for some b, along which some assertion is violated. Let b_P be the smallest bound for which such a run exists for P. Then, by definition, P is not b-correct for any b greater than b_p. Let *max-bound* B be defined as follows. $B = 1 + max\{b_P \mid P \in PG$ and P is not correct$\}$. Clearly, for every program P in PG, P is B-correct iff P is correct. The following theorem describes this observation by means of the selection vectors encoding programs in PG.

Theorem 9 (Equivalence of B-correctness and Full correctness)**.** *Let B be the max-bound defined above. Then v is an* **Rsv** *for bound B iff v is a* **Csv***. Further, v is an* **mRsv** *for bound B iff v is an* **mCsv***.*

Proof. The first part of the theorem is a direct consequence of the definition of B. The second part of the theorem is a direct consequence of the first part. This is because, by definition, v is an **mRsv** for bound B iff v is an **Rsv** for B and every v' s.t. $v' \sqsubseteq v$ and $v' \neq v$ is not an **Rsv** for B. By the first part, this happens iff v is a **Csv** and every v' s.t. $v' \sqsubseteq v$ and $v' \neq v$ is not a **Csv**, which means v is an **mCsv**.

□

Theorem 9 implies that for a large enough bound, all returned programs are correct and all minimal correct programs are returned.

6 Experimental results

We implemented a prototype of our algorithm on top of two existing tools. The translation unit and the mutation unit were implemented in C++, by modifying version 5.2 of the CBMC model checking tool [1]. The repair unit was implemented in Python, by modifying version 1.1 of the MARCO tool [16]. MARCO uses Z3 [2] as an SMT solver and Minicard [15] as a SAT solver.

Our current implementation works on C programs and uses a basic set of mutations, which is a subset of the set used in [27]. We define two *mutation levels*: level 2 contains all possible mutations and level 1 contains only a subset of them. Thus level 1 involves easier computation but may fail more often in finding a repaired program.

Table 1 shows the list of mutations used in every mutation level. For example, for the sub-category of arithmetic operator replacement, in mutation level 1, the table specifies two sets: {+,-} and {*,/,%}. This means that a + can be replaced with a - , and vice versa, and that the operators *,/,% can be replaced with each other. Constant manipulation mutations apply to a numeric constant and include increasing its value by 1 (C→C+1), decreasing it by 1 (C→C-1), setting it to 0 (C→0) and changing its sign (C→-C).

We have evaluated our algorithm on the TCAS benchmarks from the Siemens suite [7]. The TCAS program implements a traffic collision avoidance system for aircrafts. It has about 180 lines of code and it comes in 41 faulty versions, together with a reference implementation (a test suite is also included but we do not use it).

<table>
<tr><th colspan="2"></th><th>Level 1</th><th>Level 2</th></tr>
<tr><td rowspan="4">Op.
replacement</td><td>Arithmetic</td><td>{+,-},{*,/,%}</td><td>{+,-,*,/,%}</td></tr>
<tr><td>Relational</td><td>{>,>=},{<,<=}</td><td>{>,>=,<,<=},
{==,!=}</td></tr>
<tr><td>Logical</td><td colspan="2">{||,&&}</td></tr>
<tr><td>Bit-wise</td><td colspan="2">{>>, <<},{&,|,^}</td></tr>
<tr><td colspan="2">Constant
manipulation</td><td></td><td>C→C+1,C→C-1,
C→-C,C→0</td></tr>
</table>

Table 1: Partition of mutations to levels

We compared our results to those obtained by Könighofer and Bloem [11, 12]. The results are summarized in table 2. Each row refers to a different faulty version of TCAS (we only include versions for which at least one method was able to produce a repair). The specification used (in both our work and their's) is an assertion requiring equivalence with the correct version[6]. For each method there are two columns: "Fixed?", which contains a + if the method was able to find a repair for that version, and "Time", which specifies the time (in seconds) it took to find a repair (if found). The bottom line specifies for each method the number of repaired versions along with their percentage from the total 41 faulty TCAS versions, and the average time it took to find a repair.

					Our method			
	Method of [11]		Method of [12]		Mutation level 1		Mutation level 2	
Ver.	Fixed?	Time[s]	Fixed?	Time[s]	Fixed?	Time[s]	Fixed?	Time[s]
1	+	65			+	1.392	+	8.879
2	+	26	+	12				
3					+	1.725	+	68.651
6	+	55	+	79	+	2.056	+	33.762
7	+	11	+	6				
8	+	17	+	38				
9	+	41	+	28	+	1.203	+	17.286
10					+	6.429	+	90.666
12					+	2.157	+	77.852
16	+	9	+	6			+	84.711
17	+	12	+	6			+	55.538
18	+	14	+	40				
19	+	18	+	37				
20	+	85	+	26	+	1.709	+	15.883
25	+	82	+	100	+	2.68	+	16.234
28	+	34	+	35			+	93.678
31					+	1.246	+	4.661
32					+	1.902	+	85.349
35	+	41	+	46			+	92.866
36	+	8	+	6			+	94.599
39	+	82	+	101	+	2.558	+	16.393
40							+	4.829
41							+	4.875
	16 (39%)	38	15 (36.6%)	38	11 (26.83%)	2.278	18 (43.9%)	48.151

Table 2: Performance results on TCAS versions

[6] This is implemented by inlining the code of the correct version, saving the results of both versions to variables res1 and res2, and asserting that res1=res2. The code of the correct version is marked so that it will not be mutated (constraints encoding it are hard constraints).

From table 2 it is clear that there is a trade-off between repairability and runtime when deciding which mutations to use. When using mutation level 1, our method repairs less faulty versions than [11, 12] (11 vs. 15,16), but is significantly faster (2.3s vs. 38s on average). When using mutation level 2, the number of faulty versions we fix increases to 18, which is better than [11, 12], but the average time to repair increases to 48s.

For all versions that we can not repair (including those that do not appear in the table), we are able to say that they can not be fixed using the given set of mutations. Using mutation level 1 it takes approximately 2 seconds on average to reach the conclusion that the program can not be fixed using mutation sets of size 1, and approximately 7 seconds to reach that conclusion for sets of size 2 (we did not collect information about larger sizes though it is possible). Using mutation level 2 these times increase significantly to 1.5 and 24 minutes, respectively.

Note that the runtime of mutation level 1 for version number 10 is exceptionally large. This is because this version requires applying two mutations in two different locations in order to be repaired. Since we inspect programs with increasing size of mutation sets, we have to first apply all mutation sets of size 1 before inspecting any mutation sets of size 2. Though our method takes longer to produce this multi-line repair, it succeeds while [11, 12] fail.

Since the TCAS program does not contain any loops or recursive calls, all returned programs are guaranteed to be (fully) correct, and the unwinding bound is insignificant. Therefore, we also evaluated our algorithm on a set of programs with loops. This set contains implementations of commonly known algorithms (e.g., bubble-sort and max-sort) in which we inserted bugs to create different versions (a total of 10 faulty versions). All bugs can be fixed using mutation level 1, but some require multi-line repair (up to 3 mutations at a time). In all the above experiments a correct repair was found for a bound as small as 3. Furthermore, for a bound of 3, all returned programs were found to be correct (and not only bounded correct) by a manual inspection. These results suggest that though our algorithm only guarantees bounded correctness, in many cases the returned programs are correct, even when using a small bound and even in the presence of several bugs.

7 Conclusion and Future Work

This work presents a novel approach to program repair. Given an erroneous program, a set of assertions and a predefined set of mutations, our algorithm returns *all* minimal repairs to the program, in increasing number of changes.

Since the number of optional repairs might be huge, it is necessary to prune the search space whenever possible. Our technique does it by blocking all repairs that are not minimal: Whenever a successful repair is found, all repairs that use a superset of its mutations are blocked. Thus, a significant pruning of the search space is obtained.

Another promising direction is to block sets of mutations that are guaranteed *not* to succeed in repairing, based on previously seen unsuccessful once.

References

1. E. Clarke, D. Kroening, and F. Lerda. A tool for checking ansi-c programs. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, 2004.
2. L. De Moura and N. Bjørner. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.
3. L. De Moura and N. Bjørner. Satisfiability modulo theories: An appetizer. In *Formal Methods: Foundations and Applications*, pages 23–36. Springer, 2009.
4. V. Debroy and W. E. Wong. Using mutation to automatically suggest fixes for faulty programs. In *Software Testing, Verification and Validation (ICST), 2010 Third International Conference on*, pages 65–74. IEEE, 2010.
5. V. Debroy and W. E. Wong. Combining mutation and fault localization for automated program debugging. *Journal of Systems and Software*, 90:45–60, 2014.
6. F. DeMarco, J. Xuan, D. Le Berre, and M. Monperrus. Automatic repair of buggy if conditions and missing preconditions with SMT. In *Proceedings of the 6th International Workshop on Constraints in Software Testing, Verification, and Analysis*, pages 30–39. ACM, 2014.
7. H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.
8. B. Jobstmann, A. Griesmayer, and R. Bloem. Program repair as a game. In *Computer Aided Verification*, pages 226–238. Springer, 2005.
9. D. Kim, J. Nam, J. Song, and S. Kim. Automatic patch generation learned from human-written patches. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 802–811. IEEE Press, 2013.
10. E. Kneuss, M. Koukoutos, and V. Kuncak. Deductive program repair. In *Computer Aided Verification*, pages 217–233. Springer, 2015.
11. R. Könighofer and R. Bloem. Automated error localization and correction for imperative programs. In *Formal Methods in Computer-Aided Design (FMCAD), 2011*, pages 91–100. IEEE, 2011.
12. R. Könighofer and R. Bloem. Repair with on-the-fly program analysis. In *Hardware and Software: Verification and Testing*, pages 56–71. Springer, 2013.
13. C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer. A systematic study of automated program repair: Fixing 55 out of 105 bugs for 8 each. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 3–13. IEEE, 2012.
14. C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer. Genprog: A generic method for automatic software repair. *Software Engineering, IEEE Transactions on*, 38(1):54–72, 2012.
15. M. H. Liffiton and J. C. Maglalang. A cardinality solver: more expressive constraints for free. In *Theory and Applications of Satisfiability Testing–SAT 2012*, pages 485–486. Springer, 2012.
16. M. H. Liffiton, A. Previti, A. Malik, and J. Marques-Silva. Fast, flexible MUS enumeration. *Constraints*, pages 1–28, 2015.
17. M. H. Liffiton and K. A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *Journal of Automated Reasoning*, 40(1):1–33, 2008.
18. F. Long and M. Rinard. Prophet: Automatic patch generation via learning from successful patches. 2015.

19. F. Long and M. Rinard. Staged program repair with condition synthesis. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 166–178. ACM, 2015.
20. M. Martinez and M. Monperrus. Mining software repair models for reasoning on the search space of automated program fixing. *Empirical Software Engineering*, 20(1):176–205, 2015.
21. S. Mechtaev, J. Yi, and A. Roychoudhury. Directfix: Looking for simple program repairs. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1, pages 448–458. IEEE, 2015.
22. S. Mechtaev, J. Yi, and A. Roychoudhury. Angelix: Scalable multiline program patch synthesis via symbolic analysis. ICSE, 2016.
23. H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra. Semfix: Program repair via semantic analysis. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 772–781. IEEE Press, 2013.
24. Y. Pei, C. A. Furia, M. Nordio, Y. Wei, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. *Software Engineering, IEEE Transactions on*, 40(5):427–449, 2014.
25. Y. Qi, X. Mao, and Y. Lei. Efficient automated program repair through fault-recorded testing prioritization. In *2013 IEEE International Conference on Software Maintenance*, pages 180–189. IEEE, 2013.
26. Y. Qi, X. Mao, Y. Lei, Z. Dai, and C. Wang. Does genetic programming work well on automated program repair? In *Computational and Information Sciences (ICCIS), 2013 Fifth International Conference on*, pages 1875–1878. IEEE, 2013.
27. U. Repinski, H. Hantson, M. Jenihhin, J. Raik, R. Ubar, G. D. Guglielmo, G. Pravadelli, and F. Fummi. Combining dynamic slicing and mutation operators for esl correction. In *Test Symposium (ETS), 2012 17th IEEE European*, pages 1–6. IEEE, 2012.
28. S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard. Automatic error elimination by horizontal code transfer across multiple applications. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 43–54. ACM, 2015.
29. C. Von Essen and B. Jobstmann. Program repair without regret. *Formal Methods in System Design*, 47(1):26–50, 2015.
30. Y. Wei, Y. Pei, C. A. Furia, L. S. Silva, S. Buchholz, B. Meyer, and A. Zeller. Automated fixing of programs with contracts. In *Proceedings of the 19th international symposium on Software testing and analysis*, pages 61–72. ACM, 2010.
31. W. Weimer, Z. P. Fry, and S. Forrest. Leveraging program equivalence for adaptive program repair: Models and first results. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 356–366. IEEE, 2013.